Concurrent Programming with GCD in Swift 3

Session 720

Matt Wright Darwin Runtime Engineer
Pierre Habouzit Darwin Runtime Engineer
User Interface

Main Thread
User Interface
Main Thread
Data Transform
Concurrency
Concurrency

Threads allow execution of code at the same time
Concurrency

Threads allow execution of code at the same time.

CPU cores can each execute a single thread at any given time.
Concurrency

Threads allow execution of code at the same time

CPU cores can each execute a single thread at any given time

Maintaining code invariants is more difficult with concurrency
Dispatch Queues and Run Loops
Dispatch Queues and Run Loops
Dispatch Queues and Run Loops
Dispatch Queues and Run Loops
Dispatch Queues and Run Loops

Worker

Dispatch Queue
Dispatch Queues and Run Loops

Worker

Dispatch Queue

Thread
Dispatch Queues and Run Loops

Worker

Dispatch Queue

Thread

Run Loop
Dispatch Queues and Run Loops

Worker
- Dispatch Queue

Thread
- Run Loop

Main Thread
Dispatch Queues and Run Loops

Worker
- Dispatch Queue

Thread
- Run Loop

Main Thread
- Main Run Loop
- Main Queue
Asynchronous Execution
Asynchronous Execution
Asynchronous Execution

Dispatch Queue

() -> ()

() -> ()
Asynchronous Execution
Asynchronous Execution
Asynchronous Execution

Worker ➔ Dispatch Queue ➔ Worker

0 ➔ 0
0 ➔ 0
Asynchronous Execution

Worker → Dispatch Queue → ()
Asynchronous Execution

Worker

Dispatch Queue
Asynchronous Execution

Dispatch Queue
Synchronous Execution

Worker

Dispatch Queue

Thread
Synchronous Execution
Synchronous Execution

Worker

Dispatch Queue

0 -> 0

Thread

0 -> 0
Synchronous Execution

Worker

Thread

Dispatch Queue

0 -> 0
0 -> 0

() -> ()
Synchronous Execution

Worker

Dispatch Queue

Thread

() -> ()
Synchronous Execution

Worker
- Dispatch Queue
- () -> ()
- () -> ()

Thread
- () -> ()
Synchronous Execution

Worker

Thread

Dispatch Queue

0 -> 0

0 -> 0

0 -> 0
Synchronous Execution

Worker -> 0 -> 0

Thread: Dispatch Queue
Synchronous Execution
Synchronous Execution

Worker

Thread

Dispatch Queue
Synchronous Execution
Getting Work Off Your Main Thread
Getting Work Off Your Main Thread

- Transform
- User Interface
- Main Thread
Getting Work Off Your Main Thread
Getting Work Off Your Main Thread

User Interface
Main Thread
Transform
Dispatch Queue
Getting Work Off Your Main Thread

- Data
- User Interface
- Main Thread
- Transform
- Dispatch Queue
Getting Work Off Your Main Thread
Getting Work Off Your Main Thread

User Interface
Main Thread

Data
Transform
Dispatch Queue
Getting Work Off Your Main Thread
Getting Work Off Your Main Thread

Create a Dispatch Queue to which you submit work

```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)
}
```
Getting Work Off Your Main Thread

Create a Dispatch Queue to which you submit work

```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)
}
```
Getting Work Off Your Main Thread

Create a Dispatch Queue to which you submit work
Dispatch Queues execute work items in FIFO order

```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)
}
```
Getting Work Off Your Main Thread

Create a Dispatch Queue to which you submit work

Dispatch Queues execute work items in FIFO order

Use `.async` to execute your work on the queue

```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)
}
```
Getting Back to Your Main Thread

Dispatch main queue executes all items on the main thread

```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)

    DispatchQueue.main.async {
        imageView.image = smallImage
    }
}
```
Getting Back to Your Main Thread

Dispatch main queue executes all items on the main thread

```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)

    DispatchQueue.main.async {
        imageView.image = smallImage
    }
}
```
Getting Back to Your Main Thread

Dispatch main queue executes all items on the main thread
Simple to chain work between queues

```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)

    DispatchQueue.main.async {
        imageView.image = smallImage
    }
}
```
Controlling Concurrency
Controlling Concurrency

Thread pool will limit concurrency
Controlling Concurrency

Thread pool will limit concurrency
Worker threads that block can cause more to spawn
Controlling Concurrency

Thread pool will limit concurrency
Worker threads that block can cause more to spawn
Choosing the right number of queues to use is important
Controlling Concurrency

Thread pool will limit concurrency
Worker threads that block can cause more to spawn
Choosing the right number of queues to use is important
Structuring Your Application
Structuring Your Application

Identify areas of data flow in your application
Structuring Your Application

Identify areas of data flow in your application
Split into distinct subsystems
Structuring Your Application

Identify areas of data flow in your application
Split into distinct subsystems
Queues at subsystem granularity
Chaining vs. Grouping Work
Chaining vs. Grouping Work

Chaining

Grouping
Grouping Work Together

- User Interface
  - Main Queue
- Data Transform
  - Dispatch Queue
- Database
  - Dispatch Queue
- Networking
  - Dispatch Queue
Grouping Work Together

- User Interface
  - Main Queue
- Database
  - Dispatch Queue
- Data Transform
  - Dispatch Queue
- Networking
  - Dispatch Queue
Grouping Work Together

- User Interface
  - Main Queue
    - Dispatch Queue
- Data Transform
  - Dispatch Queue
- Database
  - Dispatch Queue
- Networking
  - Dispatch Queue

Dispatch Group
Grouping Work Together

```swift
let group = DispatchGroup()
```
Grouping Work Together

```
let group = DispatchQueue(group: group)
```

```
queue.async(group: group) {
...
}
```
Grouping Work Together

```
let group = DispatchGroup()
queue.async(group: group) {
    // ...}
```
Grouping Work Together

```swift
let group = DispatchGroup()

queue.async(group: group) {
    // Code
}

queue2.async(group: group) {
    // Code
}
```
Grouping Work Together

```
let group = DispatchGroup()
queue.async(group: group) {
    // code
}
queue2.async(group: group) {
    // code
}
```
Grouping Work Together

```
let group = DispatchGroup()
queue.async(group: group) { ... }
queue2.async(group: group) { ... }
queue3.async(group: group) { ... }
```

Diagram:
- User Interface
- Data Transform
- Networking
- Database
- Main Queue
- Dispatch Queue
- Dispatch Group
Grouping Work Together

```
let group = DispatchGroup()

queue.async(group: group) { ... }
queue2.async(group: group) { ... }
queue3.async(group: group) { ... }
```
Grouping Work Together

```swift
let group = DispatchGroup()

group.notify(queue: DispatchQueue.main) { … }

queue.async(group: group) { … }

queue2.async(group: group) { … }

queue3.async(group: group) { … }
```

Diagram:
- User Interface
- Database
- Networking
- Data Transform
- Main Queue
- Dispatch Queue
- Dispatch Group
Grouping Work Together

let group = DispatchGroup()

group.notify(queue: DispatchQueue.main) { ... }

queue.async(group: group) { ... }

queue2.async(group: group) { ... }

User Interface
Main Queue
Database
Dispatch Queue
Data Transform
Networking
Dispatch Queue

Dispatch Group
Grouping Work Together

```swift
let group = DispatchGroup()

group.notify(queue: DispatchQueue.main) { ... }

queue2.async(group: group) { ... }
```
Grouping Work Together

```swift
let group = DispatchGroup()

group.notify(queue: DispatchQueue.main) { … }
```
Grouping Work Together

```swift
let group = DispatchGroup()
```
Synchronizing Between Subsystems

Can use subsystem serial queues for mutual exclusion
Synchronizing Between Subsystems

Can use subsystem serial queues for mutual exclusion

Use `.sync` to safely access properties from subsystems

```swift
var count: Int {
    queue.sync {
        self.connections.count
    }
}
```
Synchronizing Between Subsystems

Can use subsystem serial queues for mutual exclusion

Use `.sync` to safely access properties from subsystems

Be aware of “lock ordering” introduced between subsystems

```swift
var count: Int {
    queue.sync {
        self.connections.count
    }
}
```
Synchronizing Between Subsystems

Can use subsystem serial queues for mutual exclusion

Use `.sync` to safely access properties from subsystems

Be aware of “lock ordering” introduced between subsystems

```swift
var count: Int {
    queue.sync { self.connections.count }
}
```
Synchronizing Between Subsystems

Can use subsystem serial queues for mutual exclusion

Use `.sync` to safely access properties from subsystems

Be aware of “lock ordering” introduced between subsystems

```swift
var count: Int {
    queue.sync { self.connections.count }
}
```
Synchronizing Between Subsystems

Can use subsystem serial queues for mutual exclusion

Use `.sync` to safely access properties from subsystems

Be aware of “lock ordering” introduced between subsystems

```swift
var count: Int {
    queue.sync { self.connections.count }
}
```
Dispatch Inside Subsystems
Choosing a Quality of Service

QoS provides explicit classification of work

- User Interactive
- User Initiated
- Utility
- Background
Choosing a Quality of Service

QoS provides explicit classification of work
Indicates developer intent

User Interactive
User Initiated
Utility
Background
Choosing a Quality of Service

QoS provides explicit classification of work
Indicates developer intent
Affects execution properties of your work

User Interactive
User Initiated
Utility
Background
Choosing a Quality of Service

QoS provides explicit classification of work
Indicates developer intent
Affects execution properties of your work

Building Responsive and Efficient Apps with GCD

WWDC 2015
Using Quality of Service Classes

```swift
queue.async(qos: .background) {
    print("Maintenance work")
}

queue.async(qos: .userInitiated) {
    print("Button tapped")
}
```
Using Quality of Service Classes

Use `.async` to submit work with a specific QoS class

```swift
queue.async(qos: .background) {
    print("Maintenance work")
}

queue.async(qos: .userInitiated) {
    print("Button tapped")
}
```
Using Quality of Service Classes

Use `.async` to submit work with a specific QoS class.

Dispatch helps resolve priority inversions.

```swift
queue.async(qos: .background) {
    print("Maintenance work")
}

queue.async(qos: .userInitiated) {
    print("Button tapped")
}
```
Using Quality of Service Classes

Use `.async` to submit work with a specific QoS class

Dispatch helps resolve priority inversions

Create single-purpose queues with a specific QoS class

```swift
queue.async(qos: .background) {
    print("Maintenance work")
}

queue.async(qos: .userInitiated) {
    print("Button tapped")
}
```
DispatchWorkItem

By default, `async` captures execution context at time of submission.
By default `.async` captures execution context at time of submission

Create `DispatchWorkItem` from closures to control execution properties

```swift
let item = DispatchWorkItem(flags: .assignCurrentContext) {
    print("Hello WWDC 2016!"
}

queue.async(execute: item)
```
DispatchWorkItem

By default, .async captures execution context at time of submission.
Create DispatchWorkItem from closures to control execution properties.
Use .assignCurrentContext to capture current QoS at time of creation.

```swift
let item = DispatchWorkItem(flags: .assignCurrentContext) {
    print("Hello WWDC 2016!")
}

queue.async(execute: item)
```
Waiting for Work Items
Waiting for Work Items

Use `.wait` on work items to signal that this item needs to execute.
Waiting for Work Items

Use `.wait` on work items to signal that this item needs to execute.

Dispatch elevates priority of queued work ahead.
Waiting for Work Items

Use `.wait` on work items to signal that this item needs to execute.

Dispatch elevates priority of queued work ahead.
Waiting for Work Items

Use `.wait` on work items to signal that this item needs to execute.

Dispatch elevates priority of queued work ahead.

Waiting with a `DispatchWorkItem` gives ownership information.

![Diagram showing main thread and queue with `.wait` signal]
Waiting for Work Items

Use `.wait` on work items to signal that this item needs to execute.

Dispatch elevates priority of queued work ahead.

Waiting with a `DispatchWorkItem` gives ownership information.

Semaphores and Groups do not admit a concept of ownership.
Shared State Synchronization

Pierre Habouzit Darwin Runtime Engineer
Swift 3 and Synchronization

Synchronization is not part of the language in Swift 3

Global variables are initialized atomically
Swift 3 and Synchronization

Synchronization is not part of the language in Swift 3

Global variables are initialized atomically
Class properties are not atomic
Swift 3 and Synchronization

Synchronization is not part of the language in Swift 3

Global variables are initialized atomically
Class properties are not atomic
Lazy properties are not initialized atomically
“There is no such thing as a benign race.”

Herb Sutter Chair of the ISO C++ standards committee
“There is no such thing as a benign race.”

Herb Sutter  Chair of the ISO C++ standards committee

Thread Sanitizer and Static Analysis  Mission  Thursday 10:00AM
Traditional C Locks in Swift

The Darwin module exposes traditional C lock types

- correct use of C struct based locks such as `pthread_mutex_t` is incredibly hard
Correct Use of Traditional Locks

`Foundation.Lock` can be used safely because it is a class
Correct Use of Traditional Locks

**Foundation.Lock** can be used safely because it is a class

Derive an Objective-C base class with struct based locks as ivars

```objective-c
@interface LockableObject :
  @implementation LockableObject {
    os_unfair_lock  _lock;
  }

- (instancetype)init ... 
- (void)lock    { os_unfair_lock_lock(&_lock); }
- (void)unlock  { os_unfair_lock_unlock(&_lock); }
@end
```
Correct Use of Traditional Locks

**Foundation.Lock** can be used safely because it is a class

Derive an Objective-C base class with struct based locks as ivars

```objective-c
@implementation LockableObject {
    os_unfair_lock _lock;
}

- (instancetype)init ...
- (void)lock   { os_unfair_lock_lock(&_lock); }
- (void)unlock { os_unfair_lock_unlock(&_lock); }
@end
```
Use GCD for Synchronization

Use `DispatchQueue.sync(execute:)`

- harder to misuse than traditional locks, more robust
- better instrumentation (Xcode, assertions, …)
// Use Explicit Synchronization

class MyObject {

    private let internalState: Int
    private let internalQueue: DispatchQueue

}

class MyObject {
    private let internalState: Int
    private let internalQueue: DispatchQueue

    var state: Int {
        get {
            return internalQueue.sync { internalState }
        }
    }
}
// Use Explicit Synchronization

class MyObject {
    private let internalState: Int
    private let internalQueue: DispatchQueue

    var state: Int {
        get {
            return internalQueue.sync { internalState }
        }
        set (newState) {
            internalQueue.sync { internalState = newState }
        }
    }
}
Preconditions

Avoid data corruption

GCD lets you express several preconditions
Preconditions

Avoid data corruption

GCD lets you express several preconditions

• Code is running on a given queue

```swift
dispatchPrecondition(.onQueue(expectedQueue)))
```
Preconditions

Avoid data corruption

GCD lets you express several preconditions

• Code is running on a given queue
• Code is not running on a given queue

```
dispatchPrecondition(.onQueue(expectedQueue)))

dispatchPrecondition(.notOnQueue(unexpectedQueue)))
```
Object Lifecycle in a Concurrent World
Object Lifecycle in a Concurrent World
Object Lifecycle in a Concurrent World

1. Single threaded setup
Object Lifecycle in a Concurrent World

1. Single threaded setup
2. **activate** the concurrent state machine
Object Lifecycle in a Concurrent World

1. Single threaded setup
2. **activate** the concurrent state machine
3. **invalidate** the concurrent state machine
Object Lifecycle in a Concurrent World

1. Single threaded setup
2. **activate** the concurrent state machine
3. **invalidate** the concurrent state machine
4. Single threaded deallocation
Object Lifecycle in a Concurrent World

1. Single threaded setup
2. activate the concurrent state machine
3. invalidate the concurrent state machine
4. Single threaded deallocation
Observer Pattern

- User Interface
  - Main Queue
- Database
  - Dispatch Queue
- Data Transform
  - Dispatch Queue
- Networking
  - Dispatch Queue
Observer Pattern

User Interface
  Main Queue

Data Transform
  Dispatch Queue
Observer Pattern

My App

User Interface
  Main Queue

Data Transform
  Dispatch Queue
Observer Pattern

class BusyController: SubsystemObserving {
    // ...
}

protocol SubsystemObserving {
    func systemStarted(...)
    func systemDone(...)
}
Observer Pattern

class BusyController: SubsystemObserving {
    // ...
}

protocol SubsystemObserving {
    func systemStarted(...)
    func systemDone(...)
}
Observer Pattern

class BusyController: SubsystemObserving {
    // ...
}

protocol SubsystemObserving {
    func systemStarted(...)
    func systemDone(...)
}
class BusyController: SubsystemObserving {

    init(...) { ... }

}
class BusyController: SubsystemObserving {
    init(...) {...}

    func activate() {
        DataTransform.sharedInstance.register(observer: self, queue: DispatchQueue.main)
    }
}
class BusyController: SubsystemObserving {
    func systemStarted(...) { /* ... */ }
    func systemDone(...) { /* ... */ }
}

My App
class BusyController: SubsystemObserving {
    deinit {
        DataTransform.sharedInstance.unregister(observer: self)
    }
}
Deallocation

BusyController

User Interface
Main Queue

Data Transform
Dispatch Queue
Deallocation

BusyController

User Interface
Main Queue

Observers
Data Transform
Dispatch Queue
Deallocation

BusyController

User Interface
Main Queue

Observers
Data Transform
Dispatch Queue
Deallocation

BusyController

User Interface
  Main Queue

Observers
  Data Transform
  Dispatch Queue

Abandoned memory
Deallocation

**Deallocation**

- Main Queue
- User Interface
- Setup
- Dispatch Queue
- Data Transform
- BusyController
- Observers

**Abandoned memory**
Deallocation

Abandoned memory
Deallocation

- Main Queue
- User Interface
- BusyController
- Observers
- Data Transform
- Octopus
- Setup
- Activated
- Invalidated
- Deallocation

Abandoned memory
Deallocation
Deallocation

- Abandoned memory
- Deadlocks
Deadlocks on Serial Queues Assert

Application Specific Information:
BUG IN CLIENT OF LIBDISPATCH: dispatch_barrier_sync called on queue already owned by current thread

Thread 1 Crashed:: Dispatch queue: com.example.queue

0 libdispatch.dylib 0x00007fff920b44ee _dispatch_barrier_sync_f_slow + 675
1 <YOUR APP> 0x000000010a3d7f26 __main_block_invoke_2 + 38
2 libdispatch.dylib 0x00007fff920a8ed6 _dispatch_client_callout + 8
3 libdispatch.dylib 0x00007fff920a9b0e _dispatch_barrier_sync_f_invoke + 83
4 <YOUR APP> 0x000000010a3d7ef6 __main_block_invoke + 38
5 libdispatch.dylib 0x00007fff920b1d54 _dispatch_call_block_and_release + 12
6 libdispatch.dylib 0x00007fff920a8ed6 _dispatch_client_callout + 8
7 libdispatch.dylib 0x00007fff920c2d34 _dispatch_queue_serial_drain + 896
...
...
Deadlocks on Serial Queues Assert

Application Specific Information:

BUG IN CLIENT OF LIBDISPATCH: dispatch_barrier_sync called on queue already owned by current thread

Thread 1 Crashed:: Dispatch queue: com.example.queue

```
0  libdispatch.dylib  0x00007fff920b44ee  _dispatch_barrier_sync_f_slow + 675
1  <YOUR APP>       0x000000010a3d7f26  __main_block_invoke_2 + 38
2  libdispatch.dylib  0x00007fff920a8ed6  _dispatch_client_callout + 8
3  libdispatch.dylib  0x00007fff920a9b0e  _dispatch_barrier_sync_f_invoke + 83
4  <YOUR APP>       0x000000010a3d7ef6  __main_block_invoke + 38
5  libdispatch.dylib  0x00007fff920b1d54  _dispatch_call_block_and_release + 12
6  libdispatch.dylib  0x00007fff920a8ed6  _dispatch_client_callout + 8
7  libdispatch.dylib  0x00007fff920c2d34  _dispatch_queue_serial_drain + 896
...```
Explicit Invalidation

class BusyController: SubsystemObserving {

  func invalidate() {
  }

  deinit {
  }
}
class BusyController: SubsystemObserving {

    func invalidate() {
        DataTransform.sharedInstance.unregister(observer: self)
    }

    deinit {
    }
}
class BusyController: SubsystemObserving {

    func invalidate() {
        dispatchPrecondition(.onQueue(DispatchQueue.main))

        DataTransform.sharedInstance.unregister(observer: self)
    }

    deinit {
    }
}
class BusyController: SubsystemObserving {

    private var invalidated: Bool = false

    func invalidate() {
        dispatchPrecondition(.onQueue(DispatchQueue.main))

        DataTransform.sharedInstance.unregister(observer: self)
    }

    deinit {
    }
}
class BusyController: SubsystemObserving {

    private var invalidated: Bool = false

    func invalidate() {
        dispatchPrecondition(.onQueue(DispatchQueue.main))
        invalidated = true
        DataTransform.sharedInstance.unregister(observer: self)
    }

    deinit {
        precondition(invalidated)
    }
}
class BusyController: SubsystemObserving {

    private var invalidated: Bool = false

    func systemStarted(...) {
        if invalidated { return }
    
        /* ... */
    }

    deinit {
        precondition(invalidated)
    }
}
GCD Object Lifecycle
let q = DispatchQueue(label: "com.example.queue", attributes: [.autoreleaseWorkItem])

let source = DispatchSource.read(fileDescriptor: fd, queue: q)
Setup

Attributes and target queue

Source handlers

```swift
let q = DispatchQueue(label: "com.example.queue", attributes: [.autoreleaseWorkItem])

let source = DispatchSource.read(fileDescriptor: fd, queue: q)

source.setEventHandler { /* handle your event here */ }
source.setCancelHandler { close(fd) }
```
Activation

Properties of dispatch objects must not be mutated after activation

```swift
extension DispatchObject {
    func activate()
}
```
Properties of dispatch objects must not be mutated after activation

```swift
extension DispatchObject {
    func activate()
}
```
Properties of dispatch objects must not be mutated after activation

- Queues can also be created inactive

```swift
extension DispatchObject {
    func activate()
}

let queue = DispatchQueue(label: "com.example.queue", attributes: [.initiallyInactive])
```
Cancellation

Sources require explicit cancellation

• Event monitoring is stopped

```swift
extension DispatchSource {
    func cancel()
}
```
Cancellation

Sources require explicit cancellation

• Event monitoring is stopped
• Cancellation handler runs

```swift
let source = DispatchSource.read(fileDescriptor: fd, queue: q)

source.setCancelHandler { close(fd) }
```
Cancellation

Sources require explicit cancellation

- Event monitoring is stopped
- Cancellation handler runs
- All handlers are deallocated

```swift
let source = DispatchSource.read(fileDescriptor: fd, queue: q)

source.setCancelHandler { close(fd) }
```
Deallocation Hygiene

GCD Objects expect to be in a defined state at deallocation

- Activated
- Not suspended
Summary

Organize your application around data flows into independent subsystems
Synchronize state with Dispatch Queues
Use the activate/invalidate pattern
More Information

https://developer.apple.com/wwdc16/720
## Related Sessions

<table>
<thead>
<tr>
<th>Session</th>
<th>Location</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Sanitizer and Static Analysis</td>
<td>Mission</td>
<td>Thursday 10:00AM</td>
</tr>
<tr>
<td>Going Server-side with Swift Open Source</td>
<td>Mission</td>
<td>Friday 9:00AM</td>
</tr>
<tr>
<td>Optimizing I/O for Performance and Battery Life</td>
<td>Nob Hill</td>
<td>Friday 11:00AM</td>
</tr>
<tr>
<td>Labs</td>
<td>Frameworks Lab D</td>
<td>Tuesday 5:00PM</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>GCD Lab</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>